Cryptographic Extraction and Key Derivation: The HKDF Scheme

Hugo Krawczyk*

Abstract

In spite of the central role of *key derivation functions (KDF)* in applied cryptography, there has been little formal work addressing the design and analysis of general multi-purpose KDFs. In practice, most KDFs (including those widely standardized) follow ad-hoc approaches that treat cryptographic hash functions as perfectly random functions. In this paper we close some gaps between theory and practice by contributing to the study and engineering of KDFs in several ways. We provide detailed rationale for the design of KDFs based on the *extract-then-expand* approach; we present the first general and rigorous definition of KDFs and their security which we base on the notion of *computational extractors*; we specify a concrete *fully practical* KDF based on the HMAC construction; and we provide an analysis of this construction based on the extraction and pseudorandom properties of HMAC. The resultant KDF design can support a large variety of KDF applications under suitable assumptions on the underlying hash function; particular attention and effort is devoted to minimizing these assumptions as much as possible for each usage scenario.

Beyond the theoretical interest in modeling KDFs, this work is intended to address two important and timely needs of cryptographic applications: (i) providing a single hash-based KDF design that can be standardized for use in multiple and diverse applications, and (ii) providing a conservative, yet efficient, design that exercises much care in the way it utilizes a cryptographic hash function.
(The HMAC-based scheme presented here, named HKDF, is being standardized by the IETF.)

*IBM T.J. Watson Research Center, Hawthorne, New York. Email: hugo@ee.technion.ac.il.
An extended abstract of this paper appears in Crypto'2010, LNCS 6223.

Contents

1 Introduction

A Key derivation function (KDF) is a basic and essential component of cryptographic systems: Its goal is to take a *source of initial keying material,* usually containing some good amount of randomness, but not distributed uniformly or for which an attacker has some partial knowledge, and derive from it one or more *cryptographically strong* secret keys. We associate the notion of "cryptographically strong" keys with that of *pseudorandom* keys, namely, indistinguishable by feasible computation from a random uniform string of the same length. In particular, knowledge of part of the bits, or keys, output by the KDF should not leak information on the other generated bits. Examples of initial keying material include the output of an imperfect physical random number generator, a bit sequence obtained by a statistical sampler (such as sampling system events or user keystrokes), system PRNGs that use renewable sources of randomness, and the less obvious case of a Diffie-Hellman value computed in a key exchange protocol.

The main difficulty in designing a KDF relates to the form of the initial keying material (which we refer to as source keying material). When this key material is given as a uniformly random or pseudorandom key K then one can use K to seed a pseudorandom function (PRF) or pseudorandom generator (PRG) to produce additional cryptographic keys. However, when the source keying material is not uniformly random or pseudorandom then the KDF needs to first "extract" from this "imperfect" source a first pseudorandom key from which further keys can be derived using a PRF. Thus, one identifies two logical modules in a KDF: a first module that takes the source keying material and extracts from it a fixed-length pseudorandom key K, and a second module that expands K into several additional pseudorandom cryptographic keys.[1]

The expansion module is standard in cryptography and can be implemented on the basis of any secure PRF. The *extraction* functionality, in turn, is well modeled by the notion of *randomness extractors* [56, 55] as studied in complexity theory and related areas (informally, an extractor maps input probability distributions with sufficient entropy into output distributions that are statistically close to uniform). However, in many cases the well-established extractors (e.g., via universal hashing) fall short of providing the security and/or functionality required in practice in the KDF context. Here we study randomness extraction from the cryptographic perspective and specifically in the context of KDFs (building upon and extending prior work [48, 23, 8, 6]). A main objective is to develop a basis for designing and analyzing secure key derivation functions following the above natural extract-then-expand approach. We are interested in the *engineering* of practical designs that can serve a variety of applications and usage scenarios and hence can be standardized for wide use. In particular, we need to be able to design extractors that will be well-suited for a large variety of sources of keying material (see Appendix B) and work in liberal as well as constrained environments. For this we resort to the use of cryptographic functions, especially cryptographic hash functions, as the basis for such multi-purpose extraction.

We identify *computational extractors*, namely randomness extractors where the output is only required to be pseudorandom rather than statistically close to uniform, as the main component for extraction in cryptographic applications, and build the notion of a KDF and its implementations on the basis of such extractors. Computational extractors are well-suited for the cryptographic setting where attackers are computationally bounded and source entropy may only exist in a com-

[1] KDF is sometimes used only with the meaning of expanding a given *strong* key into several additional keys (e.g., [58]); this ignores the extract functionality which is central to a general *multi-purpose* KDF.

putational sense. In particular, one can build such extractors in more efficient and practical ways through the use of cryptographic functions *under suitable assumptions*. Advantages of such cryptographic extractors range from purely practical considerations, such as better performance and the operational advantage of re-using functions (e.g., cryptographic hash functions) that are already available in cryptographic applications, to their more essential use for bypassing some of the inherent limitations of statistical extractors. We use the ability of cryptographic hash functions to be keyed as a way to include a *salt* value (i.e., a random but non-secret key) which is essential to obtain *generic* extractors and KDFs that can extract randomness from arbitrary sources with sufficiently high entropy.

We then study the requirements from computational extractors in the cryptographic setting, ranging from applications where the source key material has large entropy (and a good source of public randomness is also available) to the much more constrained scenarios encountered in practice where these resources (entropy and randomness) are much more limited. On this basis, we offer a KDF design that accommodates these different scenarios under suitable assumptions from the underlying cryptographic functions. In some cases, well-defined combinatorial assumptions from the hash functions will suffice while in others one has to resort to idealized modeling and "random oracle" abstractions. Our goal is to *minimize such assumptions as much as possible for each usage scenario,* but for this we need to first develop a good understanding of the properties one can expect from cryptographic hash functions as well as an understanding of the extraction functionality and the intrinsic limitations of unconditional statistical extractors in practical settings. We provide a detailed account of these issues throughout the paper.

Based on the notion of computational extractors (and on a better understanding of the complexities and subtleties of the use of KDFs in practice), we present a formal definition of the key derivation functionality suitable for capturing multiple uses of KDFs and a basis for judging the quality of general KDF designs such as those considered here. Somewhat surprisingly, in spite of being one of the most central and widely used cryptographic functionalities (in particular, specified in numerous standards), there appears to be little formal work on the specific subject of multi-purpose key derivation functions. Ours seems to be the first general definition of the KDF functionality in the literature. Our definitions include a formalization of what is meant by a "source of keying material" and they spell the security requirements from the KDF taking into account realistic adversarial capabilities such as the possession by the attacker of side information on the input to the KDF. In our formulation, KDFs accept four inputs: a sample from the source of keying material from which the KDF needs to extract cryptographic keys, a parameter defining the number of key bits to be output, an (optional) randomizing salt value as mentioned before, and a fourth "contextual information" field. The latter is an important parameter for the KDF intended to include key-related information that needs to be uniquely and cryptographically bound to the produced key material (e.g., a protocol identifier, identities of principals, timestamps, etc.).

We then use the above theoretical background, including results from [23, 17], to describe and analyze a concrete practical design of a multi-purpose KDF based on cryptographic hash functions. The scheme (denoted HKDF), that uses HMAC as the underlying mode of operation, supports multiple KDF scenarios and strives to minimize the required assumptions from the underlying hash function *for each such scenario*. For example, in some applications, assuming that the underlying hash function has simple combinatorial properties, e.g., universal hashing, will suffice while in the most constrained scenarios we will need to model the hash function as a random oracle. *The*

important point is that we will be able to use the same KDF scheme in all these cases as required for a standardized multi-purpose KDF.[2]

We end by observing that most of today's standardized KDFs (e.g., [4, 5, 57, 40]) do not differentiate between the extract and expand phases but rather combine the two in ad-hoc ways under a single cryptographic hash function (refer to Section 8 for a description and discussion of these KDF schemes). This results in ad-hoc designs that are hard to justify with formal analysis and which tend to "abuse" the hash function, requiring it to behave in an "ideally random" way even when this is not strictly necessary in most KDF applications (these deficiencies are present even in the simple case where the source of keying material is fully random). In contrast, we formulate and analyze a fully practical KDF scheme based on current theoretical research as well as on sound engineering principles. The end result is a well-defined hash-based KDF scheme applicable to a wide variety of scenarios and which exercises much care in the way it utilizes cryptographic hash functions. Our view is that given the current (healthy) skepticism about the strength of our hash functions we must strive to design schemes that use the hash function as prudently as possible. Our work is intended to fulfill this principle in the context of key derivation functions (especially at a time that new standards based on hash functions are being developed, e.g., [58]).

Related Work. As already mentioned, in spite of their importance and wide use, there is little formal work on the specific subject of multi-purpose key derivation functions. The first work to analyze KDFs in the context of cryptographic hash functions and randomness extractors appears to be [23], which was followed-up in the context of random oracles by [17]. The former work laid the formal foundations for the HMAC-based KDF scheme presented here. This scheme, in turn, is based on the KDF originally designed by this author for the IKE protocols [34, 43] and which put forth the extract-then-expand paradigm in the context of practical KDFs. A variant of the expansion stage of HKDF has also been adopted elsewhere, e.g. into TLS [20] (however, TLS does not use the extract approach; for example, keys from a DH exchange are used directly as PRF keys without any extraction operation). The extract-then-expand approach has subsequently been taken in [6] in the context of designing "system random number generators" (see Section 9); that work shares many elements with ours although the papers differ significantly in emphasis and scope. Another related work is [8] which proposes the use of statistical extractors in the design of *physical* random-number generators (see Appendix B) and points out to the potential practicality of these extractors in this specific setting. Both [6, 8] offer interesting perspectives on the use of randomness extractors in practice that complement our work; our HKDF design is well suited for use also in the settings studied by these works. A good discussion of extraction issues in the context of KDFs in the Diffie-Hellman setting can be found in [16] where a dedicated deterministic extractor for specific DH groups is presented. Another such extractor (very different in techniques and applicability) is presented in [28]. Throughout the paper we touch on additional work and we dedicate some sections in the appendix to more comparison with prior work (see the Organization paragraph below), including a critique of the traditional repeated-hash KDF scheme and its instantiation in many standards, the analysis of other hash-based KDFs, and a discussion of further applications (such as password-based KDFs and entropy-renewable random-number generators).

Organization. In Section 2 we recall the basic notions of min-entropy and extraction, and formalize computational extractors. Section 3 contains our definitions of KDF, KDF-security and related notions. Section 4 describes a generic extract-then-expand KDF and its instantiation, HKDF,

[2]The proposed HKDF scheme is being standardized by the IETF as RFC 5869.

using HMAC. In Section 5 we discuss the stringent scenarios in which one needs to resort to the random oracle modeling of the extraction functionality and study the extraction properties of random oracles. Section 6 presents the analysis of HKDF on the basis of our KDF formalization and the properties of the HMAC scheme as extractor and PRF. Section 7 presents rationale for the specific design of the expand module of HKDF. In Section 8 we compare the extract-then-expand approach (and the specific HKDF design) to the traditional repeated-hash approach prevalent in practice; this informal discussion serves to highlight the design principles and advantages of our scheme. We also discuss the shortcomings of the influential NIST's standard [57] and the work of [2]. Finally, Section 9 discusses some additional KDF applications (including password-based KDFs and entropy-renewable random-number generators).

We also include several appendices that complement and extend the material in the main body. Appendix A recalls definitions of cryptographic notions used througout the paper. Appendix B discusses in some detail several examples of sources of keying material that are common in practice and for which multi-purpose KDFs are required to work. Appendix C discusses the use of randomization in the context of extraction and KDFs, and stresses its importance and practicality as well as some of its limitations; deterministic extractors are also discussed. Appendix D contains several notes on practical aspects and variants of HKDF.

2 Statistical and Computational Extractors

This section is intended to introduce the basic notions behind the abstract randomness extraction functionality; in particular we define "computational extractors" that are central in our treatment.

The goal of the extract part of a KDF scheme is to transform the input source (seen as a probability distribution) into a close-to-uniform output. This corresponds to the functionality of *randomness extractors* which have been extensively studied in complexity theory and related areas [56]. Informally, a randomness extractor is a family of functions indexed by a public, i.e., non-secret, parameter (which we refer to as "salt") with the property that on any input distribution with sufficiently high entropy, if one chooses a salt value at random (and independently of the source distribution) the output of the extractor is statistically close to uniform (see below for a formal definition). Moreover, this statistical closeness holds even if conditioned on the salt value. Extractors with the latter property are called *strong* randomness extractors but since we only consider this type we often omit both the "strong" and "randomness" qualifiers. On the other hand, we often add the qualifier "statistical" to differentiate these extractors from computational ones (defined below) that are an essential part of our work.

Before presenting a formal definition of statistical extractors, we recall the notion of entropy considered in this context, called *min-entropy*, that captures a "worst case" notion of entropy different than the traditional average notion of Shannon's entropy (it is not hard to see that Shannon's notion is insufficient in the context of randomness extraction).

Background definitions and notation. Refer to Appendix A for some background definitions and notation used throughout the paper (e.g., the notion of δ-close).

Definition 1 *A probability distribution $\mathcal{X}$ has* min-entropy (at least) *m if for all a in the support of $\mathcal{X}$ and for random variable X drawn according to $\mathcal{X}$, $Prob(X = a) \leq 2^{-m}$.*

The following notion was first defined in [56]; see [55, 63] for surveys on the randomness extraction literature.

Definition 2 *Let $\mathcal{X}$ be a probability distribution over $\{0,1\}^n$. A function* $\mathsf{ext} : \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^{m'}$ *is called a* δ-statistical extractor *with respect to $\mathcal{X}$ if the distribution of pairs (r, y), where r is chosen with uniform probability over $\{0,1\}^t$ and $y = \mathsf{ext}_r(x)$ for x chosen according to distribution $\mathcal{X}$, is δ-close to the distribution of pairs (r, z) where z is chosen with uniform probability from $\{0,1\}^{m'}$. If* ext *is a δ-statistical extractor with respect to all distributions over $\{0,1\}^n$ with min-entropy m, then we say that* ext *is a* (m, δ)-statistical extractor.

Randomization of the extractor function via the parameter r (the salt) is mandatory if the same extractor function is to be able to extract randomness from *any* high min-entropy distribution. Indeed, for any deterministic function one can construct a high min-entropy source on which the function will produce very non-uniform outputs. On the other hand, one may consider randomness extractors that are suited for a *specific* source (or family of sources). In the latter case, one can consider *deterministic extractors*. Examples of such source-specific extractors in the cryptographic setting include the well-known hard-core schemes for RSA [3, 27] and for discrete-log based functions [38, 59], and the recent elegant extraction functions specific to some Diffie-Hellman groups in [16, 28]. For most of our study we focus on generic extractors, i.e., those that can extract randomness from *any* source with sufficient min-entropy, and hence require some non-secret salt.

A natural (and practical) question is whether common KDF applications may have a randomness source from which to obtain salt. After all, the whole purpose of extractors is to generate randomness, so if one already has such a random salt why not use it directly as a PRF key? The answer is that this randomness needs *not* be secret while in KDF applications we want the output of the extractor to be secret. Obtaining public randomness is much easier than producing secret bits, especially since in most applications the extractor key (or salt) can be used repeatedly with many (independent) samples from the same source (hence it can be chosen in an out-of-band or setup stage and be repeatedly used later). For example, a random number generator (RNG) that requires an extractor to "purify" its possibly imperfect output can simply have a random, non-secret, extractor key built-in; the same extractor key is used to purify each output from the RNG [8]. In other cases, such as key-exchange protocols extraction keys can be generated as part of the protocol (e.g., by using random nonces exchanged in the clear [34, 43]).

We further elaborate on the issue of randomization in extractors, in particular as a means to enforce independence between the source distribution and the extractor, in Appendix C.

Efficient constructions of generic (hence randomized) statistical extractors exist such as those built on the basis of universal hash functions [15]. However, in spite of their simplicity, combinatorial and algebraic constructions present significant limitations for their practical use in generic KDF applications. For example, statistical extractors require a significant difference (called the gap) between the min-entropy m of the source and the required number m' of extracted bits (in particular, no statistical extractor can achieve a statistical distance, on arbitrary sources, better than $2^{-\frac{m-m'}{2}}$ [60, 63]). That is, one can use statistical extractors (with its provable properties) only when the min-entropy of the source is significantly higher than the length of output. These conditions are met by some applications, e.g., when sampling a physical random number generator or when gathering entropy from sources such as system events or human typing (where higher min-entropy can be achieved by repeated sampling). In other cases, very notably when extracting

randomness from computational schemes such as the Diffie-Hellman key exchange, the available gap may not be sufficient (for example, when extracting 160 bits from a DH over a 192-bit group). In addition, depending on the implementation, statistical extractors may require from several hundred bits of randomness (or salt) to as many bits of salt as the number of input bits.

To obtain more practical instantiations of extractors we relax their requirements in several ways. Most significantly, we will not require that the output of the extractor be statistically close to uniform but just "computationally close", i.e., pseudorandom. The following notion is implicit in [29, 23].

Definition 3 *A* (t, ε)-computational extractor *with respect to a probability distribution* $\mathcal{X}$ *is defined as in Definition 2 except that the requirement for statistical closeness between the distributions* (r, y) *and* (r, z) *is replaced with* (t, ε)*-computational indistinguishability*[3]*. An extractor that is* (t, ε)*-computational with respect to all distributions with min-entropy* m *is called* (m, t, ε)*-computational.*

This relaxed notion will allow for more practical instantiations of extractors, particularly well-suited for the key derivation setting. Computational extractors fit the cryptographic settings where attackers are assumed to be computationally bounded, and they allow for constructions based on cryptographic hash functions. In addition, computational extraction is natural in settings such as the Diffie-Hellman protocol where the input g^{xy} to the extractor is taken from a source that has zero statistical entropy (since an attacker that knows g^x, g^y has full information to compute g^{xy}), yet may contain a significant amount of "computational min-entropy" [35] as defined next.

Definition 4 *A probability distribution* $\mathcal{X}$ *has* (t, ε)-computational min-entropy m *if there exists a distribution* $\mathcal{Y}$ *with min-entropy* m *such that* $\mathcal{X}$ *and* $\mathcal{Y}$ *are* (t, ε)*-computationally indistinguishable.*

Note: In our application of extraction to the KDF setting, where an attacker often has some a-priori information about the source (e.g., it knows the public DH values g^x, g^y from which the source key material g^{xy} is derived), we use a notion of min-entropy (statistical or computational) that is *conditioned* on such a-priori information. See Section 3.

3 Formalizing Key Derivation Functions

We present a formal definition of (secure) key derivation functions and a formalization of what is meant by a "source of keying material". To the best of our knowledge, no such general definitions have been given in the literature.[4] We start with a definition of KDF in terms of its inputs and outputs (consistent with the KDF description in Section 4). Later, after introducing the notion of sources of keying material, we define what it means for a KDF to be secure.

Definition 5 *A* key derivation function (KDF) *accepts as input four arguments: a value* σ *sampled from a source of keying material (Def. 6), a length value* ℓ*, and two additional arguments, a salt value* r *defined over a set of possible salt values and a context variable* c*, both of which are optional, i.e., can be set to the null string or to a constant. The KDF output is a string of* ℓ *bits.*[5]

[3] See Appendix A for the definition of computational indistinguishability.

[4] Yao and Yin [67] provide a formal definition of KDFs specific to the password setting which is different from and inapplicable to the general setting treated here (see Section 9).

[5] The values σ, ℓ, r, c correspond to the values *SKM*, *L*, *XTS*, *CTXinfo* in the description of Section 4.

The security and quality of a KDF depends on the properties of the "source of keying material", defined next, from which the input σ is chosen (see examples in Appendix B).

Definition 6 *A* source of keying material *(or simply* source*)* Σ *is a two-valued probability distribution* (σ, α) *generated by an efficient probabilistic algorithm.* (We will refer to both the probability distribution as well as the generating algorithm by Σ.)

This definition does not specify the input to the Σ algorithm (but see below for a discussion related to potential adversary-chosen inputs to such an algorithm). It does specify the form of the output: a pair (σ, α) where σ (the "sample") represents the (secret) source key material to be input to a KDF, while α represents some *auxiliary knowledge* about σ (or its distribution) that is available to the attacker. For example, in a Diffie-Hellman application the value σ will consist of a value g^{xy} while α could represent a quintuple (p, q, g, g^x, g^y). In a different application, say a random number generator that works by hashing samples of system events in a computer system, the value α may include some of the sampled events used to generate σ. The importance of α in our formal treatment is that we will require a KDF to be secure on inputs σ *even when the knowledge value* α *is given to the attacker.* The restriction to sources that can be generated efficiently represents our interest in sources that can arise (and be used/sampled) in practice.

Next, we define the security of a KDF with respect to a *specific* source Σ. See Definition 9 for the generic case.

Definition 7 *A key derivation function* KDF *is said to be* (t, q, ε)-secure with respect to a source of key material Σ *if no attacker* $\mathcal{A}$ *running in time* t *and making at most* q *queries can win the following distinguishing game with probability larger than* $1/2 + \varepsilon$*:*

1. *The algorithm* Σ *is invoked to produce a pair* σ, α*.*
2. *A salt value* r *is chosen at random from the set of possible salt values defined by* KDF *(*r *may be set to a constant or a null value if so defined by* KDF*).*
3. *The attacker* $\mathcal{A}$ *is provided with* α *and* r*.*
4. *For* $i = 1, \ldots, q' \leq q$*:* $\mathcal{A}$ *chooses arbitrary values* c_i, ℓ_i *and receives the value* $\mathsf{KDF}(\sigma, r, c_i, \ell_i)$ *(queries by* $\mathcal{A}$ *are adaptive, i.e., each query may depend on the responses to previous ones).*
5. $\mathcal{A}$ *chooses values* c *and* ℓ *such that* $c \notin \{c_1, \ldots, c_{q'}\}$*.*
6. *A bit* $b \in_R \{0, 1\}$ *is chosen at random. If* $b = 0$*, attacker* $\mathcal{A}$ *is provided with the output of* $\mathsf{KDF}(\sigma, r, c, \ell)$*, else* $\mathcal{A}$ *is given a random string of* ℓ *bits.*
7. *Step 4 is repeated for up to* $q - q'$ *queries (subject to the restriction* $c_i \neq c$*).*
8. $\mathcal{A}$ *outputs a bit* $b' \in \{0, 1\}$*. It wins if* $b' = b$*.*

It is imperative for the applicability of this definition that the attacker is given access to both α and r. This models the requirement that the KDF needs to remain secure even when the side-information α and salt r are known to the attacker (in particular, note that the choice of the c's and ℓ's by the attacker may depend on α and r). Allowing for multiple values of c_i to be chosen by the attacker under the same input σ to KDF ensures that even if an attacker can force the use of the same input σ to the KDF in two different contexts (represented by c), the outputs from

the KDF in these cases are computationally independent (i.e., leak no useful information on each other).

The following definition extends the min-entropy definitions from Section 2 to the setting of keying material sources (for a detailed treatment of *conditional* (computational) entropy as used in the next definition see [61, 62, 39]).

Definition 8 *We say that Σ is a* statistical m-entropy source *if for all s and a in the support of the distribution Σ, the conditional probability $Prob\,(\sigma = s \mid \alpha = a)$ induced by Σ is at most 2^{-m}. We say that Σ is a* computational m-entropy source *(or simply an m-*entropy source*) if there is a statistical m-entropy source Σ' that is computationally indistinguishable from Σ.*

We note that in the above definition we can relax the "for all a" to "all but a negligible fraction of a". That is, we can define $\alpha = a$ to be "bad" (for a given value m) if there is s such that $Prob\,(\sigma = s \mid \alpha = a) > 2^{-m}$ and require that the joint probability induced by Σ on bad a's be negligible.

Definition 9 *A KDF function is called (t, q, ε) m-*entropy secure *if it is (t, q, ε)-secure with respect to all (computational) m-entropy sources.*

We note that for the most part of this paper the (implicit) notion of security of a KDF corresponds to this last definition, namely, we think of KDFs mainly as a *generic* function that can deal with different sources as long as the source has enough computational min-entropy. We stress that this notion of security can only be achieved for randomized KDFs where the salt value r is chosen at random from a large enough set. Yet, the paper also touches (e.g., Appendix C) on deterministic KDFs that may be good for specific applications and sources and whose security is formalized in Definition 7.

On adversarially-chosen inputs to Σ. While the algorithm that generates the source Σ may have arbitrary inputs, these inputs are not provided (by our formalism) to the attacker. All auxiliary information that the attacker may have on the source Σ is modeled via the α output. But how about situations where the attacker not only has side information on the source but it can actually influence the generation of the distribution Σ? For example, an attacker can influence system events sampled by a random number generator or it can choose the parameters p, q, g (or an elliptic curve group) in a Diffie-Hellman application. Such influence could be modeled by letting the attacker choose some inputs to Σ. This, however, requires much care. First, one needs to restrict inputs only to a "legitimate set" that will ensure sufficient min-entropy in the source (e.g., disallowing $g = 1$ in a DH application). More significantly, however, is the need for independence between the source Σ and the salt value used by the KDF. Allowing the attacker to influence Σ after seeing the salt value r may result in a completely insecure KDF. As an extreme example, imagine a scenario where the algorithm Σ accepts as input a value v and then for every output σ it outputs $\alpha = \mathsf{ext}_v(\sigma)$ where ext is an extractor function that also implements the extract part of a KDF. In this case, if the attacker knows the value of the random salt r in advance (as it is the case in some applications), it could input $v = r$ into Σ and hence learn, via α, the whole output from the KDF. Thus, we have to impose careful limitations on the attacker's ability to influence the (input to) Σ source as to enforce sufficient independence with a salt value available a-priori to the attacker.

We do not pursue this formalism any further here. Instead, we refer the interested reader to [8] where the above restrictions on adversarial inputs are carefully formalized via the notion of *t-resilient extractors*. One crucial point to realize is that applications need to be designed to *enforce* sufficient independence between the source and the KDF's salt. For example, in a Diffie-Hellman protocol where honest parties choose their own salt for the KDF, this salt needs to be transmitted to the peer in an *authenticated* way or else the attacker can choose the salt itself [16]. In the example of a random number generator that fixes a non-secret random salt for its internal extractor, the application may allow some level of attacker's intervention in the generation of the source to the RNG but it must restrict it enough as to ensure the necessary level of independence with the salt value. See [8] for an extensive discussion of these issues.

4 Extract-then-Expand KDF and a HMAC-based Instantiation

In this section we first describe an abstract KDF that implements the *extract-then-expand* approach discussed throughout this paper, and then specify an instantiation solely based on HMAC [9].

4.1 A Generic Extract-then-Expand KDF

An extract-then-expand key derivation function KDF comprises two modules: a randomness extractor XTR and a variable-length output pseudorandom function PRF^* (the latter is usually built on the basis of a regular PRF with output extension via counter mode, feedback mode, etc.). The extractor XTR is assumed to produce "close-to-random", in the statistical or computational sense, outputs on inputs sampled from the source key material distribution (this should be the case also when the *SKM* value includes auxiliary knowledge α, per Definition 6, that is provided to the distinguisher). XTR may be deterministic or keyed via an optional "salt value" (i.e., a non-secret random value) that we denote by *XTS* (for *extractor salt*). The key to PRF^* is denoted by *PRK* (pseudorandom key) and in our scheme it is the output from XTR; thus, we are assuming that XTR produces outputs of the same length as the key to PRF^*. The function PRF^* also gets a length parameter indicating the number of bits to be output by the function. In all, KDF receives four inputs: the source key material *SKM*, the extractor salt *XTS* (which may be null or constant), the number L of key bits to be produced by KDF, and a "context information" string *CTXinfo* (which may be null). The latter string should include key-related information that needs to be uniquely (and cryptographically) bound to the produced key material. It may include, for example, information about the application or protocol calling the KDF, session-specific information (session identifiers, nonces, time, etc.), algorithm identifiers, parties identities, etc. The computation of the extract-then-expand KDF proceeds in two steps; the L-bit output is denoted *KM* (for "key material"):

$$\begin{aligned} 1.\ \ PRK &= \mathsf{XTR}(XTS,\ SKM) \\ 2.\ \ KM &= \mathsf{PRF}^*(PRK,\ CTXinfo,\ L) \end{aligned}$$

The following theorem establishes the security of a KDF built using this extract-then-expand approach. The proof follows from the definition of computational extractors (Definition 3), the security definition of variable-length-output pseudorandom functions (Definition 14 in Appendix A), and the definition of KDF security (Definition 7).

Theorem 1 *Let* XTR *be a* (t_X, ε_X)*-computational extractor w.r.t. a source* Σ *and* PRF* *a* $(t_P, q_P, \varepsilon_P)$*-secure variable-length-output pseudorandom function family, then the above extract-then-expand* KDF *scheme is* $(\min\{t_X, t_P\}, q_P, \varepsilon_X + \varepsilon_P)$*-secure w.r.t. source* Σ*.*

Proof: Let $\mathcal{A}_K$ be an attacker in the KDF game of Definition 7 that runs time $t_K \leq \min\{t_X, t_P\}$, makes q_K queries, and wins with probability $1/2 + \varepsilon_K$. On the basis of $\mathcal{A}_K$ we build an attacker $\mathcal{A}_X$ against XTR for source Σ as follows.

First, let's review the input-output behavior of an XTR-attacker $\mathcal{A}_X$. It receives as input three values α, r, κ that are produced as follows: source Σ is run to obtain a pair (σ, α); r is chosen at random from the set of salts to algorithm XTR; and κ is set to either $\mathsf{XTR}(r, \sigma)$ or to a random value of the same length (we will refer to these two values of κ as "real" and "random", respectively). The output of $\mathcal{A}_X$ is a bit. By the assumed security of XTR w.r.t. Σ we have that for any distinguisher $\mathcal{A}_X$ that runs time $\leq t_X$:

$$|Prob\,(\mathcal{A}_X \text{ outputs } 1 : \ \kappa \text{ is real}) - Prob\,(\mathcal{A}_X \text{ outputs } 1 : \ \kappa \text{ is random})| < \varepsilon_X \qquad (1)$$

To build an attacker $\mathcal{A}_X$ from the KDF-attacker $\mathcal{A}_K$ (see Definition 7) we proceed as follows. Upon receiving inputs α, r, κ, attacker $\mathcal{A}_X$ calls $\mathcal{A}_K$ on input α, r. For each query (c_i, ℓ_i) from $\mathcal{A}_K$, $\mathcal{A}_X$ responds with $\mathsf{PRF}^*(\kappa, c_i, \ell_i)$. For the test query (c, ℓ), $\mathcal{A}_X$ chooses a random bit b. If $b = 0$, $\mathcal{A}_X$ responds with $\mathsf{PRF}^*(\kappa, c, \ell)$, if $b = 1$ it responds with a random ℓ-bit value. When $\mathcal{A}_K$ outputs its guess b', $\mathcal{A}_X$ checks if $b = b'$ (i.e., $\mathcal{A}_K$ wins); if so, $\mathcal{A}_X$ outputs 1, else it outputs 0.

We first note that when κ is real, i.e., κ is set to the output of the extractor, then the run of $\mathcal{A}_K$ by $\mathcal{A}_X$ is a perfect simulation of a real run of $\mathcal{A}_K$ which wins the KDF game with probability $1/2 + \varepsilon_K$. On the other hand, when κ is random, then the execution of $\mathcal{A}_K$ by XTR acts as a PRF distinguisher $\mathcal{A}_P$ against PRF* with running time $t_K \leq t_P$ and $q_P = q_K$ queries, so its winning probability is of the form $1/2 + \varepsilon_P^*$ for a value $\varepsilon_P^* \leq \varepsilon_P$. We thus have that when κ is real, the probability that $\mathcal{A}_X$ outputs 1 is the same as the probability that $\mathcal{A}_K$ wins which is $1/2 + \varepsilon_K$, while for random κ the probability that $\mathcal{A}_X$ outputs 1 is $1/2 + \varepsilon_P^*$ with $\varepsilon_P^* \leq \varepsilon_P$.

To prove the theorem we need to show $\varepsilon_K < \varepsilon_P + \varepsilon_X$. In the case case that $\varepsilon_K < \varepsilon_P$ this follows trivially. In the case that $\varepsilon_K \geq \varepsilon_P$ (and then also $\varepsilon_K \geq \varepsilon_P^*$) we have, using (1):

$$\begin{aligned} \varepsilon_X &> |Prob\,(\mathcal{A}_X \text{ outputs } 1 : \ \kappa \text{ is real}) - Prob\,(\mathcal{A}_X \text{ outputs } 1: \ \kappa \text{ is random})| \\ &= |Prob\,(\mathcal{A}_K \text{ wins}) - Prob\,(\mathcal{A}_P \text{ wins})| = \\ &= |1/2 + \varepsilon_K - (1/2 + \varepsilon_P^*)| = |\varepsilon_K - \varepsilon_P^*| = \varepsilon_K - \varepsilon_P^* \geq \varepsilon_K - \varepsilon_P \end{aligned}$$

Thus, $\varepsilon_K < \varepsilon_P + \varepsilon_X$ which proves the theorem. □

4.2 HKDF: An HMAC-based Instantiation

For the sake of implementation in real applications we propose to instantiate the above general scheme with HMAC serving as the PRF underlying PRF* as well as the XTR function. We denote the resultant scheme by HKDF.

We use the following notational conventions: (i) the variable k denotes the output (and key) length of the hash function used with HMAC; (ii) we represent HMAC as a two-argument function where the first argument always represents the HMAC key; (iii) the symbol $\|$ denotes string concatenation.

Thus, when writing $\mathsf{HMAC}(a,\ b \parallel c)$ we mean the HMAC function (using a given hash function) keyed with the value a and applied to the concatenation of the strings b and c.

The scheme HKDF is specified as:

$$\mathsf{HKDF}(XTS,\ SKM,\ CTXinfo,\ L) \ = \ K(1) \parallel K(2) \parallel \ldots \parallel K(t)$$

where the values $K(i)$ are defined as follows:

$$\begin{aligned} PRK &= \mathsf{HMAC}(XTS,\ SKM) \\ K(1) &= \mathsf{HMAC}(PRK, CTXinfo \parallel 0), \\ K(i+1) &= \mathsf{HMAC}(PRK,\ K(i) \parallel CTXinfo \parallel i), \quad 1 \le i < t, \end{aligned}$$

where $t = \lceil L/k \rceil$ and the value $K(t)$ is truncated to its first $d = L \bmod k$ bits; the counter i is non-wrapping and of a given fixed size, e.g., a single byte. Note that the length of the HMAC output is the same as its key length and therefore the scheme is well defined.

When the extractor salt XTS is not provided (i.e., the extraction is deterministic) we set $XTS = 0$.

Example: Let HMAC-SHA256 be used to implement KDF. The salt XTS will either be a provided 256-bit random (but not necessarily secret) value or, if not provided, XTS will be set to 0. If the required key material consists of one AES key (128 bits) and one HMAC-SHA1 key (160 bits), then we have $L = 288$, $k = 256$, $t = 2$, $d = 32$ (i.e., we will apply HMAC-SHA256 with key PRK twice to produce 512 bits but only 288 are output by truncating the second output from HMAC to its first 32 bits). Note that the values $K(i)$ do not necessarily correspond to individual keys but they are concatenated to produce as many key bits as required.

Practical Notes. Please refer to Appendix D for several notes on the use of the HKDF in practice, including some variants such as replacing HMAC with a block-cipher based construct (e.g., CBC-MAC or OMAC) or with other "multi-property preserving" hash schemes, and using hybrid schemes where the extract and expand modules are implemented with separate components. See also Section 7 for a discussion on the use of feedback mode in the expansion stage.

Security of the HKDF scheme. See Section 6.

5 Random Oracles as Computational Extractors

In Section 2 we pointed out to the limitations of statistical extractors for use in constrained scenarios such as those with small entropy gap or limited randomness (for salt). In preparation for the security analysis in Section 6, here we study the most stringent requirements from extractors in practical KDF settings and show the role of the random oracle model in formalizing computational extraction in these demanding cases. In particular, we present results that measure the quality of random oracles as extractors. Throughout this section we use extraction from Diffie-Hellman values (computed via a key exchange, El Gamal encryption, etc.) as the main example and case study though the treatment here applies to other KDF scenarios as well (the DH case is very significant in practice and it is illustrative of the more stringent and subtle requirements from extractors)

We start by pointing out to some subtleties regarding the use of random functions in the setting of extraction. We note that random functions do not make by themselves good statistical extractors

(here the key to the extractor is the description of the function itself). For example, if $\mathcal{X}$ is the uniform distribution over $\{0,1\}^{2k}$ (thus having min-entropy $2k$) and we choose a random function f from $2k$ to k bits, we expect $1/e^2$ of the points in $\{0,1\}^k$ not to be in the image of f (i.e., they are assigned probability 0 regardless of the sample from $\mathcal{X}$), thus inducing an output distribution that is statistically far from uniform. Furthermore, the lower bounds on min-entropy gap from [60] mentioned in Section 2 apply to random functions as well. On the other hand, we will see below that when modeling functions as *random oracles* we do get good (computational) extraction properties. What makes this possible is the restriction of the attacker to a bounded number of queries to the RO (in this case, one can think of the key to the extractor as being the description of the function but the attacker is only allowed to see part of this key); this stresses the *computational* nature of the RO-based extractors.

Extraction applications served by the random oracle model. There are important applications where one has too little entropy or too little gap (i.e., the number of required output bits is close to the min-entropy of the source) to apply statistical extractors, or even non-idealized computational ones. In many of these cases, the "last resort" for analysis and validation of a design is to model the extractor component as a random oracle. Here is a list of common cases where regular extractors do not suffice (these examples are mainly from the Diffie-Hellman application but they show up in other scenarios as well).

1. A DH group where the Decisional Diffie-Hellman (DDH) is assumed but the amount of computational min-entropy in it (see [29]) is close to the number of required bits (e.g., extracting 160 bits from a 192-bit group, or even from a 160-bit group).

2. A group where the DDH assumption is not assumed or is known to fail (such as in bilinear groups). All one has is the Computational Diffie-Hellman (CDH) assumption and hence only extraction via hard core functions is possible, yet the number of required bits is larger than what can be obtained via a generic hard core such as Goldreich-Levin (see below).

3. Cases where the application does not guarantee *independence of samples*. This is the case, for example, in DH protocols where a party is allowed to re-use a DH value, or use non-independent values such as using g^x in one session and g^{x+1} in another (see discussion in [47]). It is also the case for protocols in which the attacker may be able to generate valid DH values that are dependent on previous values (this is common in "implicitly authenticated protocols", e.g., [50, 49], where DH values are input into the KDF before explicit authentication).

4. KDF (and other extraction) applications where a protocol explicitly models the KDF as a random oracle due to additional requirements from the output of this function (the previous item is such an example from the area of key-exchange protocols).

5. Cases where regular, provable extractors would work but they are not practical enough, such as for reasons of performance, amount of required randomness (salt), or simply given the lack of widespread software/hardware availability of these functions.

We stress that even in these cases, where a "super hash function" is needed to fulfill the stringent constraints of an application, one should still strive to minimize the extent of idealized assumptions. For example, hash functions that are vulnerable to extension attacks should never be modeled as random oracles; at best one should only model the compression function as random. See Section 6.

Random oracles as extractors. That random oracles may serve as good computational extrac-

tors may not be surprising. Yet, this intuition may be misleading sometimes as exemplified by the discussion on random functions at the beginning of this section. Yet, when resorting to the random oracle model in which we bound the attacker via the number of queries to the random function we can achieve meaningful extraction properties. The following lemma (adapted from [23]) shows that *(under careful restrictions)* a random oracle can output all the entropy of the source.

Lemma 2 *Let H be a random function with domain $\mathcal{D}$ and range $\{0,1\}^k$, and $\mathcal{X}$ be a probability distribution over $\mathcal{D}$ with min-entropy m and independent from H. Then the distribution $H(\mathcal{X})$ is $q2^{-m}$-indistinguishable from k random bits by any attacker that queries H on at most q points.*

Note that the independence condition in the above lemma is essential. For example, the uniform distribution on the set of points mapped by H into the all-zero strings has a large min-entropy (assuming $|\mathcal{D}| \gg 2^k$), but the output of H on this distribution is trivially (i.e., with $q = 0$) distinguishable from random. One could be tempted to make the assumption, in practice, that sources $\mathcal{X}$ of interest will be independent from the hash function used as random oracle. This, however, is unrealistic in general. Consider a DH application that uses a hash function H, say SHA-256, to hash DH keys of the form g^{xy} (i.e., it uses H as its key derivation function), but also uses H as the basis for the pseudorandom generator that generates the DH exponents x and y. In this case one cannot assume independence between the "random oracle" H and the source $\mathcal{X}$ of DH values g^{xy}.

The way to enforce independence between H and $\mathcal{X}$ is to model H as a *family* of random functions indexed by a key rather than as a single deterministic function. Then, if the KDF application chooses a function from the family H via a random key (this key, or salt, needs to be random but not secret), *and the key is independent from the source $\mathcal{X}$,* then we satisfy the independence condition from the above Lemma. Applications that cannot afford the use of salt must resort to the explicit assumption that H and $\mathcal{X}$ are independent[6](the field *CTXinfo* from HKDF can also help avoiding unwanted dependencies).

Random oracles as hard-core functions. Here we consider *extraction of pseudorandom bits from cryptographic hardness (or unpredictability) without assuming additional min-entropy,* neither computational or statistical. We use the Diffie-Hellman problem (DH) as an example. Extraction in this case is possible if the DH group satisfies the Decisional Diffie-Hellman (DDH) assumption with sufficiently high computational min-entropy (relative to the number of required pseudorandom bits) [29]. In this case, one can extract the computational min-entropy present in the value g^{xy} (conditioned on g^x, g^y) by applying to this value a statistical or computational extractor. However, when working over groups where the DDH assumption does not hold (e.g., bilinear groups), generic extractors, even computational ones, provide no guarantee that the output will be close to (or indistinguishable from) uniform; indeed, in this case there is no guaranteed computational min-entropy in g^{xy}.

One may still extract pseudorandom bits from the mere fact that it is hard to compute g^{xy} from the pair g^x, g^y (i.e., the Computational Diffie-Hellman, CDH, assumption) by resorting to the

[6]The independence assumption can be relaxed as follows. We let the attacker $\mathcal{A}$ choose any source $\mathcal{X}$ with min-entropy m but with the following restrictions: $\mathcal{A}$ is allowed to query H on q_1 values after which it sets the source distribution $\mathcal{X}$, possibly dependent on these queries. Next, x is chosen at random from $\mathcal{X}$ and $\mathcal{A}$ is given $H(x)$ or k random bits. $\mathcal{A}$ tries to decide which is the case by performing q_2 additional queries from H. The above lemma now holds with $q = q_1 + q_2$.

theory of hard-core functions [13, 32]. Roughly, given a one-way function f over $\{0,1\}^n$, we say that h is a hard core function of f if for random $x \in \{0,1\}^n$, the string $h(x)$ is pseudorandom (i.e., computationally indistinguishable from uniform) even when $f(x)$ is given (and h is known). It is well-known that every one-way function has a hard-core function, for example the Goldreich-Levin GL function [31]. This is a *generic* hard-core scheme, namely, one that works with *any* one-way function.[7] The GL function, however, has some significant limitations (some intrinsic to generic hard-core schemes) for general use in practice; in particular, it can only output a very limited number of pseudorandom bits (logarithmic in the input length), it requires a significant number of random bits as salt, it is not particularly efficient and not available in widespread implementations.

Due to the intrinsic and practical limitations of (provable) generic hard-core functions, one often resorts to specific extractors for specific applications; for example, more efficient extractors exist for the RSA and discrete-log functions [38, 37, 27]. Yet, even these extractors are not always practical enough, and certainly not general enough. In these cases, one may build on special assumptions on a given family of hash functions. In the context of DH applications, for example, this gives rise to the hashed Diffie-Hellman (HDH) assumption [1] that assumes that a specific hash function (e.g. SHA-1) is a good extractor from DH values over specific groups. However, there is little hope that one could prove anything like this for regular cryptographic hash functions such as SHA; so even if the assumption is well defined for a specific hash function and a specific group (or collection of groups), validating the assumption for standard hash functions is quite hopeless. This is even worse when requiring that a family of hash functions behaves as a *generic* extractor (i.e., suitable for arbitrary sources) as needed in a multi-purpose KDFs. In these cases, one can resort to the random oracle model as a "sanity check" for the application of hash functions as hard core functions, another example where the random oracle model is instructive for analyzing KDF schemes.

The following lemma shows, in a quantitative way, that a random oracle is as close to a generic hard-core function as one can hope. Yet, note the independence requirement in the lemma and the subsequent remark.

Lemma 3 *Assume f is a $(\tau, 2^t, \varepsilon)$-OWF, i.e., f can be computed in time τ but cannot be inverted (over random inputs) in time 2^t with probability better than ε (in case of non-permutation, inversion means finding any preimage). If H is an independent random function with k bits of output then the probability distribution $(f(x); H(x))$ over uniformly distributed x is (q, T, ε)-indistinguishable[8] from $(f(x); U_k)$ where $T = 2^t - q\tau$ and U_k denotes the uniform distribution over $\{0,1\}^k$.*

The proof follows standard arguments and is omitted.

Remark: Lemma 3 assumes that f is independent of H, namely, computing f does not involve queries to H. This assumption can be relaxed and f be allowed to depend on H as long as computing $f(x)$ (as well as sampling x) does not involve computing H on x. On the other hand, some independence assumption on the choice of H is necessary as *deterministic* hard-core functions do not exist[9] even if implemented via a random function.

Random Oracle advisory. To conclude, the random-oracle model may be invoked in the context of KDF design as a "sanity check" for the use of hash functions as specific or generic extractors;

[7] The CDH problem is not strictly a one-way function, yet this technical difficulty can be lifted as shown in [65].

[8] That is, no distinguisher running in time T and making q queries to H has distinguishing advantage better than ε.

[9] If f is a one-way function and h is a deterministic hard-core function then f' defined as $f'(x) = (f(x), h(x))$ is also one-way but, obviously, h is not a hard-core function for f'.

e.g., as a way to model the hashed DH assumption without entering the specific details of a group or a hash function. As with any application of the random-oracle model [26, 12] one has to be careful about the way one interprets results obtained in this model. The main usefulness of this model is in uncovering possible *structural weaknesses* in a cryptographic design but it ultimately says little about the actual security of a specific real-world design where one uses a hash function instead of the random oracle. Thus, while we warn against the widespread practice of building KDF schemes *solely* on the basis of abstracting hash functions as random functions, we do not ignore the potential benefits of this model as a verification tool, especially when we do not have a better way to validate a design or its application in specific scenarios.

6 The Security of HKDF

Theorem 1 from Section 4 allows us to argue the security of HKDF on the basis of the properties of the HMAC scheme both as extractor and PRF. In this section we review results concerning these properties of HMAC and use them to prove the security of HKDF. These results demonstrate that the structure of HMAC works well in achieving the basic functionalities that underline HKDF including PRF, extraction, and random-oracle domain extension. In particular, they exploit the versatility of HMAC that supports working with a secret key, a random non-secret key (salt), or deterministically (i.e., with a fixed-value key). We also note that the security analysis of HKDF uses in an essential way the structure of HMAC and *would not hold* if one simply replaces HMAC with a plain (Merkle-Damgard) hash function.

Notation. We use H to denote a Merkle-Damgard hash function and h the underlying compression function. We also consider these as keyed families, where h_κ and H_κ denote, respectively, the compression function and the Merkle-Damgard hash with their respective IVs set to κ; the key and output lengths of these functions is denoted by k. We will abuse notation and talk about "the family h_κ" instead of the more correct $\{h_\kappa\}_{\kappa\in\{0,1\}^k}$; same for H_κ. When we say that "the family h_κ is random", we mean that each of the functions h_κ is chosen at random (with the corresponding input/output lengths). When we talk about HMAC (or NMAC), we assume underlying functions h and H (or their keyed versions).

The properties of HMAC as a pseudorandom function family are well established [10, 11] and are based on the assumed pseudorandomness of the underlying compression function family h_κ.[10] It is not hard to see that the use of HMAC in"feedback mode" in HKDF (for realizing PRF*) results in a secure variable-length-output pseudorandom function family. Indeed, the latter is a generic transformation from fixed-length output PRF into a variable-length output PRF* (see more on this transformation and the rationale for the use of feedback mode in Section 7).

The suitability of HMAC as a computational extractor is more complex and is treated in detail below. These results show the extraction properties of HMAC for a wide variety of scenarios under suitable assumptions on the underlying hash function, ranging from purely combinatorial properties, such as universality, to the idealized modeling of compression functions as random oracles.

We first state the following general theorem.

[10] Although the security of HMAC as PRF degrades quadratically with the number of queries, such attack would require the computation of the PRF (by the owner of the key) over inputs totalizing $2^{k/2}$ blocks. This is not a concern in typical KDF applications where the number of applications of the PRF is relatively small.

Theorem 4 *(informal) Let H be a Merkle-Damgard hash function built on a family of pseudo-random compression functions $\{h_\kappa\}_\kappa$. Let $\mathcal{S}$ be a collection of probability distributions acting as sources of keying material. Assume that the instantiation of HMAC with the family $\{h_\kappa\}_\kappa$ is a secure computational extractor w.r.t. sources in $\mathcal{S}$, then* HKDF *is a secure KDF w.r.t. sources in $\mathcal{S}$.*

The theorem follows from Theorem 1 applied to the collection of sources $\mathcal{S}$ and the fact, discussed above, that HMAC is a secure PRF when instantiated with a pseudorandom family of compression functions h_κ. *Each of the lemmas presented below provides a condition on HMAC extraction that can be plugged into this theorem to obtain a proof of security for* HKDF *for the appropriate sources of key material in a* well-defined and quantifiable *way.*

The results below involve the notion of "almost universal (AU)" hash functions [15, 66]: A family h_κ is δ-AU if for any inputs $x \neq y$ and for random κ, $Prob(h_\kappa(x) = h_\kappa(y)) \leq \delta$. This is a natural (combinatorial) property of hash functions and also one that any (even mildly) collision resistant hash family must have and then a suitable assumption for cryptographic hash functions. Specifically, if the hash family h_κ is δ-collision-resistant against linear-size circuits (i.e., such an attacker finds collisions in h_κ with probability at most δ) then h_κ is δ-AU [23]. For results that apply to the most constrained scenarios (as those discussed in Section 5) we need to resort to stronger, idealized assumptions, in which we model functions as *random oracles (RO)*, namely, random functions which the attacker can only query on a limited number, q, of queries.

NMAC as extractor. The following lemmas are adapted from [23] and apply directly to the NMAC scheme underlying HMAC (recall that $\mathsf{NMAC}_{\kappa_1,\kappa_2}(x) = H_{\kappa_2}(H_{\kappa_1}(x))$, where H_{κ_2} is called the "outer function" and H_{κ_1} the "inner function"). The results extend to HMAC as explained below. They show that HMAC has a structure that supports its use as a generic extractor and, in particular, it offers a much better design for extraction than the plain hash function H used in many of the existing KDFs.

Lemma 5 *If the outer function is modeled as a RO and the inner function is δ-AU then NMAC applied to an m-entropy source produces an output that is $\sqrt{q(2^{-m}+\delta)}$-close to uniform where q is a bound on the number of RO queries.*

The above modeling of the outer function as a random oracle applies to the case where the outer function is a single (fixed) random function (in which case the source distribution needs to be independent of this function) or when it is represented as a keyed family of random functions (in which case only the key, or salt, needs to be chosen independently of the source distribution).

One natural question is whether one can ensure good extraction properties for NMAC based on the extraction properties of the underlying compression functions and *without idealized assumptions.* The following result from [23] provides an affirmative answer for *m-blockwise sources*, namely, where each k-bit input block has min-entropy m when conditioned on other blocks. Denote by $\hat{h}_\kappa$ a family identical to the compression function family h_κ but where the roles of key and input are swapped relative to the definition of h_κ.

Lemma 6 *If h_κ is a (m,δ)-statistical extractor and $\hat{h}_\kappa$ is a (t,q,ε)-pseudorandom family for $q=1$, then NMAC is a $(t, n\delta+\varepsilon)$-computational extractor for m-blockwise sources with n input blocks.*

An example of a practical application where this non-idealized result can be used is the IKE protocol [34, 43] where all the defined "mod p" DH groups have the required block-wise (computational) min-entropy. In particular, the output from HKDF is guaranteed to be pseudorandom in this case without having to model h as a random oracle.

Truncated NMAC. Stronger results can be achieved if one truncates the output of NMAC by c bits to obtain $k' = k - c$ bits of output (e.g., one computes NMAC with SHA-512 but only outputs 256 bits). In this case one can show that NMAC is a good *statistical* extractor (not just computational) under the following sets of assumptions:

Lemma 7 *If h_κ is a family of random compression functions (with k bits of output) then NMAC truncated by c bits is a $(k, \sqrt{(n+2)2^{-c}})$-statistical extractor where n is the number of input blocks.*

Lemma 8 *If the inner function is δ_1-AU and the outer is $(2^{-k'} + \delta_2)$-AU then truncated NMAC (with k' bits of output) is a $(m, \sqrt{2^{k'}(2^{-m} + \delta_1 + \delta_2)})$-statistical extractor.*

The latter lemma is particularly interesting as its guarantee is "unconditional": it does not depend on hardness assumptions or idealized assumptions, and it ensures statistical extraction. Moreover, it fits perfectly the HKDF setting if one implements the extract phase with, say, SHA2-512 (with output truncated to 256 bits) and the PRF part with SHA2-256 (as discussed in Appendix D). In particular, we have the following significant case:

Corollary 9 *If the family of compression functions h_κ is strongly universal (or pairwise independent) and the family H_κ is generically collision resistant against linear-size circuits, then NMAC truncated by c bits is a $(k, (n+2)2^{-c/2})$-statistical extractor on n-block inputs.*

Indeed, the assumption that the family h_κ is strongly universal means $\delta_2 = 0$; and it is not hard to see that if there is no trivial (linear size) algorithm to find collisions in the family H_κ better than guessing then $\delta_1 \leq n2^{-k}$. Putting these values and $m = k$ into Lemma 8 the corollary follows.

Applying the corollary to the above example of SHA2-512 truncated to 256 bits we get a statistical distance of $(n+2)2^{-128}$. And there is plenty room to get good security even if the δ values deviate from the above numbers; e.g., for $\delta_1 = \delta_2 = 2^{100}2^{-k}$ we would get a statistical closeness of $(n+2)2^{-78}$. Finally, we note that a min-entropy of $m = k$ is a condition satisfied by many common distributions such as statistical samplers and DH groups modulo safe primes.

From NMAC to HMAC. To use the above results with HMAC one needs to assume the computational independence of the values $h(\kappa \oplus \text{opad})$ and $h(\kappa \oplus \text{ipad})$ for random κ (i.e., each of these values is indistinguishable from uniform even if the other is given). In the cases where h is modeled as a random function this requires no additional assumption.

HMAC as a random oracle. In Section 5 we point out to various scenarios that require the modeling of the extraction functionality through random oracles. This may be due to the stringent requirements of an application (e.g., when all of the min-entropy of the source is to be extracted), when extraction can be solely based on cryptographic hardness without assuming additional min-entropy (the "hard core" case), or when the application itself assumes the KDF to be a random oracle (as in certain key exchange protocols). In all these cases we are interested to model the

extract part of HKDF as a random oracle. Fortunately, as shown in [17] (using the framework of indifferentiability from [51]), the HMAC structure preserves randomness in the sense that if the underlying compression function (computed on fixed length inputs) is modeled as a random oracle so is HMAC on variable length inputs ([17] claims the result for a variant of HMAC but it applies to HMAC itself).[11] This result together with Lemma 2 from Section 5 implies:

Lemma 10 *If the compression function h is modeled as a RO, then the application of HMAC to an m-entropy source produces an output that is $q2^{-m}$-close to uniform where q is a bound on the number of RO queries.*

As explained in Section 5, the above holds for source distributions that are (sufficiently) *independent* from the function h. To obtain a generic extractor one needs to randomize the scheme by keying HMAC with a salt value.

Finally, we point out that using Lemma 3 from Section 5, one obtains that if h_κ is a RO family then HMAC over the family H_κ is a generic hard-core family. This is needed when extraction is to be based on cryptographic hardness (or unpredictability) only without assuming additional min-entropy (e.g., in a Diffie-Hellman exchange where only CDH is assumed – see Section 5).

7 The Key-Expansion Module

Implementing the expand functionality of a KDF is relatively straightforward as it can be based on any pseudorandom function (with the right input size parameters)[12]. The most basic PRFs in practice, such as a block cipher or a keyed compression function, are defined with fixed-length input and fixed-length output. To transform these PRFs so they can accept arbitrary-length inputs one resorts to common modes of operations such as CBC-MAC or OMAC for block ciphers and Merkle-Damgard for compression functions [53, 18].

For the KDF application, we also need the PRF to support variable-length outputs, hence we need to define a way to iterate an (arbitrary-input) PRF to produce outputs of any required length. The simplest and most common solution for this is to use "counter mode", namely, computing the PRF on successive values of a counter. Since PRFs are defined and designed to produce independent pseudorandom outputs even from dependent inputs, this mode is perfectly fine with strong PRFs. In our design of HKDF we take a more conservative approach[13] via "feedback mode" where rather than computing the PRF on successive counter values, each iteration is applied to the result of previous iteration.

Our rationale for this choice is mostly heuristic (a formal analysis can be carried via the notion of *weak pseudorandom functions* [54]). In feedback mode the attacker has less knowledge on the inputs to the PRF, except for the first input which is fully known (as in counter mode). At a formal

[11] This "RO preserving" property does not hold for the plain Merkle-Damgard hash which is susceptible to extension attacks. Moreover, even if one considers fixed-length inputs (to avoid extension attacks), the Merkle-Damgard family H_κ built on random compression functions is not a good statistical extractor (e.g., [23] show that the output of such family on any distribution for which the last block of input is fixed is statistically far from uniform).

[12] Note that, in principle, one could use a pseudorandom generator (PRG) for this purpose but a PRG does not accommodate the *CTXinfo* field and hence we recommend against it.

[13] The strength and long-term security of the PRF are important since its cryptanalysis could compromise past communications and information protected under keys derived using the PRF.

level this may not be an advantage since the output of PRF* should be unpredictable even when the attacker has knowledge of all bits except one; however, in practical terms PRF* is likely to present a significantly harder cryptanalytical problem. This includes a smaller amount of known plaintext than counter mode (or even chosen plaintext if the attacker can force some counter values). Another good example of the advantage of highly variable inputs (even if they were known to the attacker) is that successive values of a counter differ in very few bits which creates relationships such as low Hamming differentials between inputs, a potential aid for cryptanalysis.

Note that we also concatenate a counter value to each input to the PRF intended to force different inputs as a defense against potential cycle shortcuts [64].

8 Traditional vs Extract-then-Expand KDFs

One way to provide further rationale for the design of HKDF is to compare between the traditional approach to the design of KDFs (as encountered in most standards and real-world applications) and the extract-then-expand methodology underlying the HKDF design. We present such an informal comparison here, and then discuss a particular and influential instantiation of the traditional approach in the NIST standard [57]. We end with an analysis of a scheme from [2].

A comparison between traditional KDFs and HKDF. The traditional KDFs (e.g., [4, 5, 40]) can be described schematically as:

$$KM = \mathsf{Hash}(SKM \parallel \text{“1”} \parallel info) \parallel \mathsf{Hash}(SKM \parallel \text{“2”} \parallel info) \parallel \ldots \parallel \mathsf{Hash}(SKM \parallel \text{“}t\text{”} \parallel info)$$

where Hash is implemented with a *plain* hash function such as SHA-1; *SKM* denotes the source key material, “*i*” denotes a fixed-length counter, $\parallel$ denotes concatenation, and *info* describes (optional) context- and application-specific information. The value t depends on the amount of generated keying material, denoted *KM*, required by the application.

This approach *does not differentiate between the extract and expand phases;* rather it combines the two in ad-hoc ways under a single cryptographic hash function (often thought of as an ideal random function), resulting in a *repeated* application of the hash function on the source key material. This repetition imposes stronger requirements from the hash function relative to the extract-then-expand design in which the hash function is applied only once to the *SKM* value to “distill” all its entropy into a single cryptographic key (which is then used to key a PRF for additional key material). In the traditional approach the hash function needs to be able to extract many more bits from the same *SKM*. Moreover, the different inputs to Hash are strongly dependent. Even the best information-theoretic extractors fail to extract so many bits of randomness and even less so under *highly correlated inputs*. In contrast, in the HKDF design, the dependent inputs go into the PRF since PRFs are *specifically designed* to remain secure even when the inputs (except for the key) are fully known and correlated, or even adversarially chosen.[14]

One essential (and minimal) test for the appropriateness of a design as a KDF is to check how it fares when the input *SKM* is already a strong cryptographic key (e.g., a secret random value such as in the TLS protocol). In this case, the functionality of the KDF is identical to that of

[14]Even in this case, HKDF minimizes correlation via the use of feedback mode (rather than counting mode) that avoids low input variability (e.g., low-Hamming differentials) and adds unpredictability to the PRF inputs.

a pseudorandom function and therefore we should expect the KDF to implement a strong PRF in this case. For random *SKM*, HKDF becomes an application of HMAC which is the standard and widely accepted PRF mode for Merkle-Damgard hash functions. The traditional scheme, in contrast, builds the PRF using the hash function in "key-prepended mode", a mode that has long been deprecated in favor of schemes such as HMAC. In particular, key-prepended schemes need to care about extension attacks thus requiring careful prefix-free encoding of the input. Moreover, such a scheme does not even follow the usual practice of aligning (via padding) the prepended key to a block boundary (as to induce a random IV).

The implication of these considerations is that the only way to justify the traditional KDF design in terms of cryptographic analysis is to model the plain hash function as a random oracle. In contrast, while we sometimes resort to the random oracle modeling for analyzing HKDF in the most constrained scenarios, in many other (practical) instances the requirements we put on the underlying hash function are significantly more relaxed (even when modeled as a random oracle the HMAC scheme is superior to the plain hash function as it does not suffer from extension attacks and can be modeled on the basis of the compression functions being random).

Yet another significant drawback of the traditional schemes is that they do not take advantage of a random salt even if available. As demonstrated by our analysis, the use of such random salt significantly strengthens the extraction functionality and hence the overall KDF design.

To conclude, the structure of the extract-then-expand approach is much better suited for KDF design than the repeated hash, in particular by recognizing the very different functionalities behind the extract and the expand stages. Luckily we are able to implement both components using HMAC which, in particular, has significant advantages as a building block over the use of the plain hash function (e.g., the results backing HKDF in Section 6 apply to HMAC and not to the plain underlying hash function). Overall, the results in this paper demonstrate that the structure of HMAC works well in achieving the basic functionalities that underline HKDF including PRF, extraction, and random-oracle domain extension. This is achieved thanks to the versatility of HMAC that supports working with a secret key, a random non-secret key (salt), or deterministically (i.e., with a fixed-value key).

NIST's variant of the traditional scheme. NIST's KDF is defined in [57] and intended for the derivation of keys from a secret shared via a key-exchange protocol. The scheme is a variant of the "traditional KDF scheme" discussed above. The difference is in the ordering of the value *SKM* and the counter which are swapped relative to the traditional scheme, namely, NIST's scheme is:

$$KM = \mathsf{Hash}(\text{“1”} \parallel SKM \parallel info) \parallel \mathsf{Hash}(\text{“2”} \parallel SKM \parallel info) \parallel \ldots \parallel \mathsf{Hash}(\text{“}t\text{”} \parallel SKM \parallel info)$$

One can wonder about the reasons for this deviation from the traditional scheme; unfortunately, NIST documents do not offer such rationale. Clearly, this change does not help against the problems we identified in the traditional scheme. To the contrary, when NIST's KDF is examined as a PRF (namely, when the key material *SKM* is strong enough as a PRF key) it results in a *very non-standard* PRF that inherits weaknesses from both the "key-prepend mode" (i.e., $\mathsf{Hash}(K \parallel \cdot)$) and the "key-append mode" (i.e., $\mathsf{Hash}(\cdot \parallel K)$).[15]

[15]The use of a prepended counter has been discussed in [2] as a possible defense against the "2^k attack" that we describe below. However, NIST's own document [19] correctly points out that their KDF scheme is not claimed or intended to provide security beyond 2^k.

Another peculiarity of NIST's KDF is that the document ([57], Sec. 5.8) explicitly forbids (with a strong **shall not**) the use of the KDF for generating non-secret randomness, such as IVs. If there is a good reason to forbid this, namely the KDF is insecure with such a use, then there must be something very wrong about this function (or, at least, it shows little confidence in the strength of the scheme). First, it is virtually impossible to keep general applications and implementors from using the KDF to derive auxiliary random material such as IVs. Second, if the leakage of an output from the KDF, in this case the public IV, can compromise other (secret) keys output by the KDF, then the scheme is fully broken; indeed, it is an essential requirement that the leakage of one key produced by the KDF should not compromise other such keys.[16]

Finally, NIST's KDF is "over-specified" in that it mandates specific information to be used in the *info* field such as parties identities. This makes the KDF specific to key-exchange (KE) protocols (this may be a deliberate choice of NIST but not one that helps on standardizing a multi-purpose KDF). Moreover, this specification excludes perfectly fine KE protocols, such as IKE, where party identities are not necessarily known when the KDF is called. (NIST mandated the use of identities for reasons that are specific to the KE protocols defined in [57] and not as a general principle applicable to *all* sound KE protocols.)

Adams et al paper [2]. One of the few works in the literature that we are aware of that treats specifically the design of generic (multi-purpose) KDFs is the work of Adams et al [2]. While not formal in nature, the paper is instructive and includes a comprehensive description of many KDF standards. The main theme in the paper is a critique of existing schemes on the following basis (this applies to the traditional schemes as well as ours). Due to the way the hash function is used by these schemes, their strength (measured by the work factor it takes to guess the output of the KDF) is never more than 2^k where k is the output length of the hash function, and this is the case even when the entropy of the source key material is (much) larger than k. Indeed, in these schemes it suffices to guess $\mathsf{Hash}(SKM)$ to be able to compute all the derived key material. While we agree with this fact, we do not agree with the criticism. If a work factor of 2^k is considered feasible (say 2^{160}) then one should simply use a stronger hash function. Counting on a 2^k-secure hash function to provide more than 2^k security, even on a high-entropy source, seems as unnecessary as unrealistic.

Even the very scheme from [2] claimed to overcome the 2^k limitation does not achieve this goal. Specifically, [2] states the following as an explicit goal of a KDF: If the entropy of the source of key material is m and the number of bits output by the KDF and not provided to the attacker is m', then the work factor to learn these hidden m' output bits should be at least $2^{\min\{m,m'\}}$. In particular, in the case of a hash-based KDF, this work factor should be independent of the output length of the hash function. The scheme proposed in [2] is the following. On input source key material SKM the KDF outputs the concatenation of $K(1), \ldots, K(t)$ where $K(1) = H(S); K(i) = H_{K(i-1)}(S),\ i = 2, \ldots, t$, (here H_K denotes the hash function H with the IV set to K). In Appendix E we show, however, that this scheme does not achieve the intended goal, by presenting an attack that retrieves the full KDF output in much less than the bounds set as the scheme's security.

Therefore, we repeat, if a 2^k work factor is deemed feasible, then go for a stronger hash function. It is better to focus on the right structure of the KDF and to avoid the multiple extraction from the same SKM value which is common to the traditional scheme as well as to the scheme from [2].

[16] It seems that NIST's spec assumes another source of public randomness from which IVs and similar values are derived. If such a source is available it should be used to salt the KDF as we recommend here.

9 Other Related Work and Applications

IKE. As we mentioned earlier, the extract-then-expand approach to KDF design was first adopted (at least in the setting of real-world protocols) with our design of the KDF of the IKE protocol (IPsec's Key Exchange) [34, 48]. The scheme presented here follows that design. What has changed substantially since the original design of IKE's KDF is the understanding of the theoretical foundations for this design. The present work is intended to stress the virtues of this design as a generic multi-purpose KDF, and to present in detail (and in a formal way) the design rationale based on this new understanding.

Password-based KDF. One specific area where KDFs are used is in deriving cryptographic keys from passwords. For these password-based KDFs (PBKDF) the goal is to derive a key from a password in such a way that dictionary attacks against the derived key will not be practical. For this, typical PBKDFs (e.g., PKCS #5 [44]) use two main ingredients: (i) salt to prevent building universal dictionaries, and (ii) the slowing down of the KDF operation via repeated hash computations (the latter technique referred as "key stretching" in [45]). This makes PBKDFs very different than the general-purpose KDFs studied here. In particular, while passwords can be modeled as a source of keying material, this source has too little entropy to meaningfully apply our extractor approach except when modeling the hash function as a random oracle. Also the slowing-down approach of PBKDFs is undesirable for non-password settings. Our two-stage KDF approach from Section 4 could still be used to build PBKDFs by replacing the extraction stage with a key-stretching operation (implemented via some hash repetition technique), or by interleaving such key stretching between the extraction and expansion phases. In the latter case, one would first derive an intermediate key P from the password pwd and salt XTS as $P = \mathsf{HMAC}(XTS, pwd)$, then apply key stretching to P to obtain a key PRK, and finally expand PRK to the number of required cryptographic keys via PRF^*. The interested reader should consult [67] for a formal treatment of PBKDFs and [45] for the properties required from key-stretching techniques and their underlying hash functions. We point out that another formalism that may help bridging between our extraction approach and PBKDFs is the notion of *perfectly one-way probabilistic hash functions* from [14].

Applications to System RNGs. KDFs can be used as a basis for a system "random number generator" (RNG) which typically consists of some sampling process used to "extract and renew" randomness from some source in order to seed a pseudorandom generator (PRG). The first work to treat this problem formally is [6] which also proposes the use of the extract-then-expand approach to the construction of such generators. This work bears many common elements with ours although the (multi-purpose) KDF functionality presents a wider range of applications and scenarios, and hence requires a more comprehensive treatment which we offer here. On the other hand, [6] provides a much more detailed and formal study of the specific system RNG application (taking care of randomness refreshing, dealing with partial leakage of internal state, etc.). We note that our KDF design, specifically our HMAC-based instantiation HKDF, is well suited for the above application as well, and would result in a much sounder RNG than many of the ad-hoc constructions used in practice (c.f., [24]). Another related work is [8] which proposes the use of combinatorial extractors in the design of *physical* random-number generators (Appendix B) and points out to the potential practicality of these extractors in this setting. Both [6, 8] offer interesting perspectives on the use of randomness extractors in practice that complement our work.

A Background Definitions

In this section we recall basic formal definitions for some of the notions used throughout this work.

Notation. In the sequel $\mathcal{X}$ and $\mathcal{Y}$ denote two (arbitrary) probability distributions over a common support set A; X and Y denote random variables drawn from $\mathcal{X}$ and $\mathcal{Y}$, respectively.

Definition 10 *We say that probability distributions $\mathcal{X}$ and $\mathcal{Y}$ have* statistical distance *δ (or are δ-*close*) if $\sum_{a \in A} |Prob(X = a) - Prob(Y = a)| \leq \delta$.*

Definition 11 *An algorithm D is an ε-*distinguisher *between distributions $\mathcal{X}$ and $\mathcal{Y}$ if $|Prob(D(X) = 1) - Prob(D(Y) = 1)| < \varepsilon$.*

We note that two distributions $\mathcal{X}$ and $\mathcal{Y}$ are δ-close iff there is no ε-distinguisher between $\mathcal{X}$ and $\mathcal{Y}$ for $\varepsilon > \delta$.

By restricting the computational power of the distinguisher in the above definition one obtains the following well-known definition of "computational indistinguishability" [33] (we formulate the definition using the "concrete security" (t, ε) approach as a non-asymptotic alternative to the classical polynomial-time treatment).

Definition 12 *Two probability distributions $\mathcal{X}$, $\mathcal{Y}$ are (t, ε)-*computationally indistinguishable *if there is no ε-distinguisher between $\mathcal{X}$ and $\mathcal{Y}$ that runs in time t.*

Definition 13 *A probability distribution $\mathcal{X}$ over the set $\{0,1\}^n$ is called (t, ε)-*pseudorandom *if it is (t, ε)-computationally indistinguishable from the uniform distribution over $\{0,1\}^n$.*

Next we recall the definition of security for a variable-length output pseudorandom function family. Such a family consists of a collection of keyed functions which on input a key κ, an input c and a length parameter ℓ, outputs ℓ bits.

Definition 14 *A variable-length output pseudorandom function family PRF^* is (t, q, ε)-*secure *if no attacker $\mathcal{A}$ running in time t and making at most q queries can win the following distinguishing game with probability larger than $1/2 + \varepsilon$:*

1. *For $i = 1, \ldots, q' \leq q$: $\mathcal{A}$ chooses arbitrary values c_i, ℓ_i and receives the value $\mathsf{PRF}^*(\kappa, c_i, \ell_i)$ (queries by $\mathcal{A}$ are adaptive, i.e., each query may depend on the responses to previous ones).*
2. *$\mathcal{A}$ chooses values c and ℓ such that $c \notin \{c_1, \ldots, c_q\}$.*
3. *A bit $b \in_R \{0,1\}$ is chosen at random. If $b = 0$, attacker $\mathcal{A}$ is provided with the output of $\mathsf{PRF}^*(\kappa, c, \ell)$, else $\mathcal{A}$ is given a random string of ℓ bits.*
4. *Step 4 is repeated for up to $q - q'$ queries.*
5. *$\mathcal{A}$ outputs a bit $b' \in \{0,1\}$. It wins if $b' = b$.*

Notes on terminology. In our presentation in this paper we often use the above terms without explicit mention of the (t, ε) parameters (e.g, we talk about computational indistinguishability – or just indistinguishability – rather than the more accurate (t, ε)-computational indistinguishability). In these cases it is understood that the missing t is an amount of infeasible computation (for the attacker) while ε is typically a small or even negligible quantity. More generally, the ratio t/ε is assumed to be larger than the expected running time of any feasible attacker.

B Sources of Imperfect Keying Material

The extract part of a KDF depends on the type of initial keying material, the available entropy, the number of bits to output, etc. Here we discuss several examples of sources of (imperfect) initial keying material for KDFs as encountered in important practical applications. In particular, this serves to show the multi-functionality of KDFs and the challenge in creating a *single* KDF scheme that can deal with all these cases.

Physical RNG. Consider a physical random number generator RNG. Such generator often produces outputs that are not perfectly uniform but do have sufficient min-entropy (a simple example is a generator that outputs 0 with fixed probability $p < 1/2$ but not less than, say, $1/4$). To produce a truly (close to) uniform output one uses an extractor on a (sufficiently long) sequence generated by the RNG.[17] We note that in this application one can salt (or key) the extractor with a non-secret random value that is chosen once (from some off-line randomness-producing process) and fixed as an RNG parameter. See [8] for more on the use of extractors in the context of physical RNGs.

Ad-hoc statistical sampling. Consider the typical practice of sampling system events, clocks, user keystrokes, etc., to produce an initial source of randomness for seeding a PRG or other randomness applications (especially when a source of physical randomness is not present). This application is based on the assumption that the sampled source contains sufficient variability such that it is partially unpredictable by an external attacker. In other words, the source is assumed to behave as a probability distribution with sufficient min-entropy conditioned on information held by the attacker (which can see part of the events in a system or even partially influence them). Here again, applying an extractor on a (long enough) sequence of such samples produces a close-to-uniform output. As in the RNG example, one can set a random extractor key at some initial stage and use this salt repeatedly with multiple samples. We note that some (operating) system RNGs use "renewable sources" of entropy, by resorting to periodic statistical sampling as a way to keep seeding the RNG. This case is studied in detail in [6] and is further discussed in Section 9.

Key Exchange and Diffie-Hellman Protocols. Diffie-Hellman protocols output random group elements of the form g^{xy} represented by strings of some length n. Clearly, these values are not uniformly distributed over $\{0,1\}^n$ and therefore cannot be used directly as cryptographic keys. In turn, keys are usually derived from g^{xy} via a KDF under the assumption that g^{xy} contains "sufficient randomness" that an attacker cannot predict. More precisely, when modeled appropriately, the

[17] In this example if one samples n bits from RNG one gets that the most probably string is the all-1 string which has probability at most $(3/4)^n$, i.e. the source has min-entropy $-\log_2(3/4)n \approx 0.4n$. For example, if $n = 1000$ one can use a statistical extractor and output 200 bits with a statistical deviation from uniform of at most 2^{-100}.

values g^{xy} and their probability distribution, as implied by the DH protocol, become a particular case of an "imperfect source of randomness". One essential difference with previous examples is that in the Diffie-Hellman setting the notion of min-entropy is *computational* (note that an attacker that knows g^x, g^y has full information to compute g^{xy}, and therefore the statistical entropy of g^{xy} given g^x, g^y is zero.) Fortunately, the computational min-entropy of a source is sufficient for deriving *pseudorandom* bits via regular randomness extractors as considered here (see Section 2).

While this generation of pseudorandomness via statistical extraction from DH values applies to many protocols and DH groups, there are important cases where this does not hold. For example, not all Diffie-Hellman groups offer "sufficient" computational min-entropy for extraction purposes. In particular, this can only be guaranteed (see [29]) in groups where the Decisional Diffie-Hellman (DDH) assumption holds in a sufficiently large subgroup. This is not the case, for example, in DH groups based on pairings where the DDH assumption does not hold. In addition, in order to apply a randomness extractor in the setting of DH key-exchange protocols, it is necessary that the value g^{xy} computed by the parties be indistinguishable (by the attacker) from a random group element. This is the case for some DH protocols but not for all. For example, in a protocol such as MQV (or its provable variant HMQV) [50, 49] one can argue that the shared group element computed by the protocol is hard to fully guess by an attacker but not necessarily hard to distinguish from random group elements. Finally, even when one can use the DDH assumption one may not have enough min-entropy to extract the required number of bits (e.g., when requiring the extraction of 160 bits of key from a DH value over a 192-bit prime order elliptic curve). In all these cases, one cannot use regular generic extractors but rather needs to resort to stronger properties or assumptions as discussed in Section 5.

One additional consideration when using randomness extractors in the setting of DH protocols is that these extractors work well only on *independent samples*. That is, if one uses the extractor to generate a key out of a value g^{xy} and later to generate another key from a second value $g^{x'y'}$, the two keys will be pseudorandom and independent *provided that the two DH values are independent*. In particular, an attacker should not be able to force dependencies between DH values exchanged between honest parties. This is the case in a (well designed) DH protocol with explicit authentication. In protocols with implicit authentication or those that authenticate the protocol via the output of a KDF, one may not obtain the independence guarantee. This is yet another case where stronger extractors are required (see more in Section 5).

We end by commenting that one good property for the use of extractors in Diffie-Hellman applications is that the parties to the protocol can provide the necessary "salt" to key the extractor. Indeed, such value can be obtained from random nonces chosen by the parties and exchanged authenticated but *in the clear*. This is the approach we took in the design of IKE [34, 43, 48].

C Randomized vs. Deterministic Extractors

As we have already stressed in previous sections *generic* extractors, i.e., those working over arbitrary high min-entropy sources, must be randomized via a random, but not necessarily secret, key (or "salt"). In particular, the example following Lemma 2 shows that for every deterministic extractor there is a high-entropy source for which the output is far from uniform.

In general, randomization provides a way to *enforce independence between the source distribution and the extractor itself*, and between different uses of the same extractor scheme by an application.

The importance of such independence can be illustrated by the following "real-life" example. The Photuris protocol, a predecessor of today's IKE protocol, used a variant [52] of the STS protocol where a Diffie-Hellman exchange is authenticated by signing the key g^{xy}. In actuality, what is signed is $H(g^{xy})$ for some hash function H, say SHA-1. If the signature allows for message recovery (e.g. RSA) then the attacker does not learn g^{xy} but rather $H(g^{xy})$; thus if one thinks of H as a random oracle then nothing is learned about g^{xy}. Now, suppose that the above protocol derives keys via a traditional KDF (Section 8), namely, keys are computed as $H(g^{xy} \parallel v)$ for some public values v. In this case, if we follow the H =SHA-1 example and assume g^{xy} is of length 1024, we get that computing $H(g^{xy} \parallel v)$ is the same as computing H on v with the IV of H set to $H(g^{xy})$ (for simplicity assume that the computation of H on g^{xy} does not use length appending). However, this means that the attacker learns all the keys since he knows $H(g^{xy})$! The failure of the protocol is not due to a specific weakness of H but rather to the use of the same (deterministic) function for two different functionalities: the hash function applied to g^{xy} and the KDF. In contrast, if one keyed H via a salt value in its use by the KDF one would enforce independence between the two functionalities.

The role of independence, in a more technical sense, is also highlighted by the results presented in Section 5; there we also discuss the necessity of randomness in cases where one uses hard-core functions instead of extractors. We also comment that using random keys with extractors has additional advantages, at least at the heuristic level, such as avoiding potentially weak functions in a hash family and, importantly, preventing attacks based on pre-processing that targets a specific function.

A valid question is whether typical KDF applications can use randomization (or salt). As we argued in Section 2 (and exemplified in Appendix B) this is indeed the case in many important cases (including RNGs, key exchange, etc.). Yet, when such salt is not available one has to resort to *deterministic extractors*. These require careful source-specific assumptions as well as independence assumptions on how an application uses the extractor. Note that the Photuris failure would happen also if one built on the "hash Diffie-Hellman (HDH) assumption" [1] where one assumes that a specific hash function, say SHA-1, acts as a good extractor from a specific group (or family of groups). Another option when randomization is not possible is to resort to extractors that work in particular cases and hence of restricted applicability. Examples of such "domain-specific" extractors include the hard-core schemes for RSA [3, 27] and discrete-log based functions [38, 59], and the elegant extraction functions specific to some Diffie-Hellman groups in [16, 28].

An interesting direction for further investigation is the use of deterministic extraction when one is provided with two independent randomness sources; see [7] (interestingly, the extractor from [16] may be seen as a specific case of the above approach).

On secret salt. See Appendix D.

D Some Practical Notes on HKDF

We present several notes related to the use and variants of HKDF in practice (they complement our description of HKDF from Section 4).

1. In the case in which the source key material *SKM* is already a random (or pseudorandom) string of the length of the PRF^* key, there is no need to extract from it further randomness and one can

simply use *SKM* as the key to PRF*. Also, when *SKM* is from a source or form that requires the use of the extraction step but the number of key-material bits required is no larger than the output *PRK* of the extractor, one could use directly this key as the output of the KDF. Yet, in this case one could still apply the PRF* part as an additional level of "smoothing" of the output of XTR. In particular, this can help against potential attacks on XTR since by applying PRF* we make sure that the raw output from XTR is never directly exposed.

2. Applications that use HKDF need to specify the contents of *CTXinfo*. As said this is useful (and sometimes essential) to bind the derived key material to application- and context-specific information, so the information in *CTXinfo* needs to be chosen judiciously. On the other hand, a general specification as ours, does not define any particular format or value for *CTXinfo* (contrast this to NIST's KDF [57] discussed in Section 8). We note that *CTXinfo* should be independent of *SKM*; more precisely, the entropy of the source conditioned on *CTXinfo* should be sufficiently high (see Section 3). For example. in a DH application, *CTXinfo* should not include $SKM = g^{xy}$ but could include g^x, g^y.

3. Applications may consider inputting the length L as a parameter to PRF*. This is necessary only if an application may be calling the KDF twice with the same *SKM* and the same *CTXinfo* field but with different values L, and still expect the outputs to be computationally independent of each other. While possible, we do not see this as a usual scenario and hence prefer not to mandate it; specific applications can chose to do this either as an additional parameter to the KDF or as part of the *CTXinfo* field. (We do not have any important application in mind for which inputting the length would be a problem but we prefer to be more liberal here as to avoid usability problems in the future – see the discussion on the "over-specification" of NIST's KDF in Section 8.)

4. The generic extract-then-expand specification can be instantiated with other than HMAC functions. For example, one could instantiate XTR with HMAC but PRF* based on AES in CBC-MAC (or OMAC [42]) or using any secure variable-length output PRF such as those listed in [58]. In cases where *CTXinfo* is null, PRF* can be replaced with a pseudorandom generator seeded with *PRK*.

Per results from [23], the extract module can also be based on a block-cipher using CBC-MAC mode (with *XTS* as the key); however the analytical results in this case are much more restricted than with HMAC. The extraction part can also use a dedicated extractor tailored to a specific application/source (e.g., Diffie-Hellman values over some group family); we discuss source specific extractors in Appendix C. Finally, we point out to a particularly advantageous instantiation using HMAC-SHA512 as XTR and HMAC-SHA256 in PRF* (in which case the output from SHA-512 is truncated to 256 bits). This makes sense in two ways: First, the extraction part is where we need a stronger hash function due to the unconventional demand from the hash function in the extraction setting. Second, as shown in Section 6, using HMAC with a truncated output as an extractor allows to prove the security of HKDF under considerably weaker assumptions on the underlying hash function.

Note. The generic KDF specification implicitly assumes that the output from XTR is no shorter than the key to PRF*. This is always the case in HKDF that uses the same function for both XTR and PRF*, but it could be an issue, for example, if one implements XTR with SHA-256 and PRF* with HMAC-512. In this case (which we do not recommend since we want to put the stronger primitive on the more demanding XTR part) one needs to define a larger output from XTR. If and how this can be done depends on the specific extractor function in use (e.g., if the extractor

function is also designed as a good PRF, as in the HMAC case, one could use a scheme similar to PRF* to obtain more bits from XTR).

5. If one modifies the value $K(1)$ in the definition of HKDF to $K(1) = \mathsf{HMAC}(PRK, 0 \parallel CTXinfo \parallel 0)$, where the first 0 field is of length k, one ensures that all inputs to HMAC for generating the $K(i)$ values are of the same length. This is not strictly needed with HMAC (which, for example, is not vulnerable to extension attacks) but it may be advisable with other PRF modes (such as CBC-MAC), and it is often a good basis for better analytical results. We have not included this 0 field in our definition of HKDF for compatibility with the IKE's KDF [34, 43] which does not use it.

6. On "secret salt": Some applications may have available a *secret* key K and still need to extract another key from a source such as a Diffie-Hellman value. This is the case in IKE (public-key encryption and preshared modes) [34, 43] and TLS's preshared key mode [25]. For this, it is best to use the secret K as a key to a PRF applied to *SKM*. HKDF is particularly well-suited for this case since by setting *XTS* to the secret key K one obtains exactly the PRF effect (historically, this dual use of salt has been one of the motivations for our design of the IKE's KDF).

E Attack on the KDF Scheme from [2]

As stated in Section 8, the scheme from Adams et al. [2] does not achieve the security goal set for that scheme by its authors. We build a counter-example using the following scenario. The hash block size is 512 bits and the source S consists of strings of length 1024 and has entropy m (for simplicity of presentation we assume that in the computations of $K(i)$ the length appending of typical hash functions is omitted; the argument holds also with length appending). There are numbers m_1, m_2, each smaller than 512, such that $m = m_1 + m_2$, and $m = 2k$. The first half of S is taken uniformly from a set M_1 of size 2^{m_1} and the second half is taken uniformly from a set M_2 of size 2^{m_2}, and both M_1 and M_2 can be efficiently enumerated (e.g., M_1 is the set of all strings with $512 - m_1$ least significant bits set to 0 and the rest chosen uniformly).

We now mount the following "meet in the middle" attack. Say the output of the KDF is $K_1, \ldots, K_t$ for $t \geq 4$ (we are using notation K_i instead of $K(i)$) and we are given K_1 and K_2 (i.e., $m' \geq 2k$). We find the value of S by running the following procedure, and from S we compute the remaining values $K_3, K_4, ...$

1. For each $S_2 \in M_2$ and $K \in \{0,1\}^k$, check if $H_K(S_2) = K_1$. Create a table T for all pairs (K, S_2) passing the check.
2. For each $S_1 \in M_1$ check if the pair $(H(S_1), S_2)$ is in T for some S_2. If so check if $H_{K_1}(S_1||S_2) = K_2$. If both checks are satisfied output $S = S_1||S_2$. If no such pair is found return 'fail'.

First, note that if the above procedure returns S then we have that $H(S) = K_1$ and $H_{K_1}(S) = K_2$. Moreover, since we know that such S exists (K_1, K_2 are given to us as the output produced by such a value S) then the procedure never fails. The question is whether the (first) value found is the correct S. Now, assuming a random H, the number of values S expected to produce a given K_1 and a given K_2 is $2^m/2^{2k}$. Since we choose $m = 2k$ then this value is 1 and hence we find the correct S with high probability. Clearly, once we found S we can compute the remaining K_i's.

The complexity of the above procedure is as follows. We need time 2^{k+m_2} for step 1. To help in step 2 we need to sort the table T. Since the expected size of T is 2^{m_2} (we go through 2^{k+m_2}

values each with probability 2^{-k} to hit the value K_1) then the sorting work is negligible relative to 2^{k+m_2}. Step 2 itself takes 2^{m_1} checks where each check entitles a table lookup into T which takes $O(\log |T|) = O(m_2)$. In all the work is order of $2^{k+m_2} + 2^{m_1} m_2$. Putting $m_1 = 1.5k$ and $m_2 = 0.5k$ we get that the whole work takes approximately $2^{1.5k}$ which is much less than the 2^{2k} time that [2] intended the work of finding K_3 and K_4 (and beyond) to take.

Note 1: if we assume that the block K_2 is used by a specific algorithm then we do not even need to learn K_2 to mount the above attack but only have a way to test it (e.g., if K_2 is used to key an encryption algorithm then we need plaintext-ciphertext pairs computed under that key).

Note 2: If we replace H with HMAC in Adams we have the same meet-in-the-middle attack; even if salted with *XTS*. The attack is also possible against our scheme but we do not claim security beyond 2^k.

References

[1] M. Abdalla, M. Bellare, and P. Rogaway, "The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES", *CT-RSA '01*, LNCS No. 2020., pages 143–158, 2001.

[2] Carlisle Adams, Guenther Kramer, Serge Mister and Robert Zuccherato, "On The Security of Key Derivation Functions", ISC'2004, LNCS 3225, 134-145.

[3] Werner Alexi, Benny Chor, Oded Goldreich, Claus-Peter Schnorr: RSA and Rabin Functions: Certain Parts are as Hard as the Whole. SIAM J. Comput. 17(2): 194-209 (1988)

[4] ANSI X9.42-2001: Public Key Cryptography For The Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography.

[5] ANSI X9.63-2002: Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport.

[6] Boaz Barak, Shai Halevi: A model and architecture for pseudo-random generation with applications to /dev/random. ACM Conference on Computer and Communications Security 2005.

[7] Boaz Barak, Russell Impagliazzo, and Avi Wigderson, "Extracting randomness from few independent sources", FOCS 04.

[8] Boaz Barak, Ronen Shaltiel, and Eran Tromer, "True random number generators secure in a changing environment", Cryptographic Hardware and Embedded Systems (CHES), LNCS no. 2779, 2003.

[9] M. Bellare, R. Canetti, and H. Krawczyk. "Keying hash functions for message authentication", *Crypto '96*, LNCS No. 1109. pages 1–15, 1996.

[10] M. Bellare, R. Canetti, and H. Krawczyk. "Pseudorandom Functions Revisited: The Cascade Construction and Its Concrete Security", *Proc. 37th FOCS*, pages 514–523. IEEE, 1996.

[11] Mihir Bellare, "New Proofs for NMAC and HMAC : Security Without Collision-Resistance", CRYPTO 2006, LNCS 4117. pp. 602-619.

[12] Mihir Bellare, Phillip Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols", ACM Conference on Computer and Communications Security 1993.

[13] Manuel Blum, Silvio Micali: How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. SIAM J. Comput. 13(4): 850-864 (1984)

[14] R. Canetti, D. Micciancio, and O. Reingold, "Perfectly one-way probabilistic hash functions", Proc. STOC'98, pp. 131-140.

[15] L. Carter and M. N. Wegman. "Universal Classes of Hash Functions", *JCSS*, 18(2), 1979.

[16] Olivier Chevassut, Pierre-Alain Fouque, Pierrick Gaudry, David Pointcheval: The Twist-AUgmented Technique for Key Exchange. Public Key Cryptography 2006: LNCS 3958.

[17] Jean-Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, Prashant Puniya: "Merkle-Damgard Revisited: How to Construct a Hash Function", CRYPTO'05, LNCS 3621, 430-448.

[18] I. Damgard. A Design Principle for Hash Functions. *Crypto '89*, LNCS No. 435, pages 416–427.

[19] Q. Dang and T. Polk, "Hash-Based Key Derivation (HKD)", draft-dang-nistkdf-01.txt (work in progress), June 23, 2006.

[20] T. Dierks and C. Allen, ed., "The TLS Protocol – Version 1", *Request for Comments 2246*, 1999.

[21] T. Dierks and E. Rescorla, ed., "The TLS Protocol – Version 1.1", *Request for Comments 4346*, 2006.

[22] W. Diffie, P. van Oorschot and M. Wiener, "Authentication and authenticated key exchanges", *Designs, Codes and Cryptography*, 2, 1992, pp. 107–125

[23] Dodis, Y., Gennaro, R., Håstad, J., Krawczyk H., and Rabin, T., "Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes", Crypto'04, LNCS 3152.

[24] Leo Dorrendorf, Zvi Gutterman and Benny Pinkas, "Cryptanalysis of the Windows Random Number Generator", ACM Computer and Communications Security Conference (CCS'07), 2007.

[25] P. Eronen and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, Dec. 2005.

[26] A. Fiat, A. Shamir, "How to Prove Yourself: Practical Solutions to Identification and Signature Problems", CRYPTO 1986: 186-194.

[27] R. Fischlin, C.-P. Schnorr, "Stronger Security Proofs for RSA and Rabin Bits", Eurocrypt'97.

[28] P.-A. Fouque, D. Pointcheval, J. Stern, S. Zimmer, "Hardness of Distinguishing the MSB or LSB of Secret Keys in Diffie-Hellman Schemes", ICALP (2) 2006: LNCS 4052.

[29] R. Gennaro, H. Krawczyk and T. Rabin. "Secure Hashed Diffie-Hellman over Non-DDH Groups", Eurocrypt'04.

[30] Rosario Gennaro, Luca Trevisan: Lower Bounds on the Efficiency of Generic Cryptographic Constructions. FOCS 2000: 305-313.

[31] Oded Goldreich, Leonid A. Levin: A Hard-Core Predicate for all One-Way Functions. STOC 1989.

[32] Oded Goldreich, *Foundations of Cryptography*. Cambridge University Press, 2001.

[33] S. Goldwasser and S. Micali, "Probabilistic Encryption", *JCSS*, 28(2):270–299, April 1984.

[34] D. Harkins and D. Carrel, ed., "The Internet Key Exchange (IKE)", *RFC 2409*, Nov. 1998.

[35] J. Hastad, R. Impagliazzo, L. Levin, and M. Luby. "Construction of a Pseudo-random Generator from any One-way Function", *SIAM. J. Computing*, 28(4):1364–1396, 1999.

[36] Johan Hstad, Mats Naslund: Practical Construction and Analysis of Pseudo-Randomness Primitives. ASIACRYPT 2001: 442-459.

[37] Johan Håstad, Mats Naslund: The security of all RSA and discrete log bits. J. ACM 51(2): 187-230 (2004).

[38] J. Hastad, A. Schrift, A. Shamir, "The Discrete Logarithm Modulo a Composite Hides O(n) Bits," *J. Comput. Syst. Sci.*, 47(3): 376-404 (1993)

[39] Chun-Yuan Hsiao, Chi-Jen Lu, Leonid Reyzin, "Conditional Computational Entropy, or Toward Separating Pseudoentropy from Compressibility", EUROCRYPT 2007, pp. 169-186

[40] IEEE P1363A: Standard Specifications for Public Key Cryptography: Additional Techniques, Institute of Electrical and Electronics Engineers.

[41] I. Impagliazzo and D. Zuckerman, "How to Recycle Random Bits", in Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science, pp. 248–253, 1989.

[42] Tetsu Iwata and Kaoru Kurosawa, "OMAC: One-Key CBC MAC", FSE 2003: 129-153.

[43] C. Kaufman, ed., "Internet Key Exchange (IKEv2) Protocol", RFC 4306, Dec. 2005.

[44] B. Kaliski, PKCS #5: Password-Based Cryptography Specification Version 2.0, RFC 2898, Sept. 2000.

[45] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure Applications of Low-Entropy Keys", *ISW 1997*, LNCS 1396, pp. 121-134.

[46] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, Feb. 1997.

[47] Krawczyk, H., "SKEME: A Versatile Secure Key Exchange Mechanism for Internet", *1996 Internet Society Symposium on Network and Distributed System Security*, pp. 114–127.

[48] H. Krawczyk. "SIGMA: The 'SiGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols", *Crypto '03*, pages 400–425, 2003

[49] Hugo Krawczyk: HMQV: A High-Performance Secure Diffie-Hellman Protocol. CRYPTO 2005. LNCS 3621: 546-566.

[50] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone, "An efficient Protocol for Authenticated Key Agreement", *Designs, Codes and Cryptography, 28, 119-134, 2003.*

[51] Ueli M. Maurer, Renato Renner, Clemens Holenstein: Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. TCC 2004: 21-39.

[52] A. Menezes, P. Van Oorschot and S. Vanstone, "Handbook of Applied Cryptography," CRC Press, 1996.

[53] Ralph C. Merkle: One Way Hash Functions and DES. CRYPTO 1989: 428-446.

[54] Moni Naor, Omer Reingold: Synthesizers and Their Application to the Parallel Construction of Pseudo-Random Functions. J. Comput. Syst. Sci. 58(2): 336-375 (1999).

[55] N. Nisan and A. Ta-Shma. "Extracting Randomness: A Survey and New Constructions", *JCSS*, 58:148–173, 1999.

[56] N. Nisan and D. Zuckerman. "Randomness is linear in space", *J. Comput. Syst. Sci.*, 52(1):43–52, 1996.

[57] NIST Special Publication (SP) 800-56A, Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography, March 2006.

[58] NIST Special Publication (SP) 800-108, Recommendation for Key Derivation Using Pseudorandom Functions. October 2009. `http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf`

[59] S. Patel and G. Sundaram. "An Efficient Discrete Log Pseudo Random Generator", *Crypto '98*, LNCS No. 1462, pages 304–317, 1998.

[60] Jaikumar Radhakrishnan and Amnon Ta-Shma. "Tight bounds for depth-two superconcentrators," SIAM J. Discrete Math. 13(1): 2-24 (2000).

[61] R. Renner and S. Wolf. "Smooth Renyi entropy and applications." Proceedings of IEEE International Symposium on Information Theory, 2004.

[62] R. Renner and S. Wolf. "Simple and tight bounds for information reconciliation and privacy amplification." ASIACRYPT 2005.

[63] R. Shaltiel. "Recent developments in Extractors", Bulletin of the European Association for Theoretical Computer Science, Volume 77, June 2002, pages 67-95. Available at: http://www.wisdom.weizmann.ac.il/~ronens/papers/survey.ps

[64] Adi Shamir and Boaz Tsaban, "Guaranteeing the Diversity of Number Generators," *Information and Computation*, Vol. 171, Issue 2, 2001, pp. 350-363

[65] V. Shoup, "Lower Bounds for Discrete Logarithms and Related Problems", Eurocrypt'97, LNCS 1233, pp. 256-266

[66] Douglas R. Stinson: Universal Hashing and Authentication Codes. Des. Codes Cryptography 4(4): 369-380 (1994).

[67] Frances F. Yao, Yiqun Lisa Yin, "Design and Analysis of Password-Based Key Derivation Functions," CT-RSA 2005.